View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Composing Programming Languages by Combining Action-Semantics Modules

Kyung-Goo Doh [1,2]

Department of Computer Science and Engineering
Hanyang University, Ansan, South Korea

Peter D. Mosses [3,4]

BRICS & Department of Computer Science
University of Aarhus, Denmark

Abstract

This article demonstrates a method for composing a programming language by combining action-semantics modules. Each module is defined separately, and then a new module is defined by either extending or combining existing modules. This method enables the language designer to gradually develop a language by selecting, extending and combining suitable language modules. The resulting modular structure is substantially different from that previously employed in action-semantic descriptions.

We also discuss how to resolve the conflicts that may arise when combining modules, and indicate some advantages that action semantics has over other approaches in this respect.

1 Introduction

Hoare noted in his POPL'73 paper [7,8] that much of programming language design is consolidation, not innovation. In other words, a language designer should largely utilize constructions and principles that have worked well in earlier design projects and experiments [21]. To this end, language definitions should be modularized, so that the language constructs whose concepts and

[1] This work was supported by grant No.2000-1-30300-010-3 from the Basic Research Program of the Korea Science & Engineering Foundation.
[2] Email: doh@cse.hanyang.ac.kr
[3] Supported by BRICS: Basic Research in Computer Science, Centre of the Danish National Research Foundation.
[4] Email: pdmosses@brics.dk

©2001 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.

operations are closely related are collected into a separate module. Then, a language designer's job may involve the definition or reuse of many different alternative language modules, to choose the best combination of them based on language-design principles, and to reject any that show mutual inconsistencies. Any remaining minor inconsistencies or overlaps may be reconciled by applying good engineering principles. In order to facilitate the process of defining and extending language-definition modules, a framework must provide a high degree of modularity so that the designer can build up a language smoothly and uniformly. For defining (concrete or abstract) context-free syntax, BNF grammars have excellent modularity; for defining semantics, the framework of *action semantics* appears to offer similar advantages.

Action semantics has been developed by Mosses and Watt [13,14,15,17,25]. The motto of action semantics is to allow *useful* semantic description of *realistic* programming languages [13]. In action semantics, a high-level notation called *action notation* is used to describe the meaning of a programming language. Primitive actions exist for the fundamental behaviours of information processing: value passing, arithmetic, binding creation and lookup, storage allocation and manipulation, and so on. Actions are composed into a combined action with combinators controlling the flow of information. Actions have semi-descriptive English names, which gives a novice hints of the intuition behind them. Furthermore, action semantics specifications are inherently *modular*. Hence, they can be straightforwardly extended and modified to reflect language design changes, and to reuse parts of an existing language definition for specifying similar, related languages. In fact, some real-world programming languages, such as Pascal [18], and Standard ML [26,27], have been successfully described in action semantics.

1.1 This work

This article demonstrates a method for composing a programming language by combining action-semantics modules. A language module is the collection of language constructs whose concepts and operations are closely related [22]. Each module is defined separately, and then a new module is defined by either extending or combining existing modules. This method enables the language designer to gradually develop a language by selecting, extending and combining suitable language modules. The modules are generally much smaller than those previously employed in action-semantic descriptions, and their overall organization is significantly different. It appears that the adoption of the novel style of module proposed here should greatly facilitate the direct reuse of entire modules in action semantics.

When combining modules, conflicts may occur in the definition of a combined module. This article suggests how to resolve the problem of conflicts, exploiting several key features of the action semantics framework. Modules for constructs stemming from the so-called Landin-Tennent design principles

[21] are particularly uniform, and straightforward to combine.

Note that only dynamic semantics is considered in this article since the main focus here is to illustrate the good modularity and extensibility of action semantics. Static semantics (e.g., type-checking semantics) can be defined in inference-rule format as in Plotkin's lecture notes on SOS [19] and in Schmidt's textbook on typed programming languages [21], or in action notation as in Watt's work [26,27], becoming the essential part of a language specification. Static semantics can be automatically extracted from dynamic action semantics if the language to be defined is statically typed [4,5].

1.2 Road map

The rest of the article is organized as follows: Section 2 briefly introduces action semantics and defines base modules and extension modules necessary to compose an expression language. Section 3 presents modules for expression bindings, expression blocks, expression parameters, and expressions with effects. Section 4 shows how a language module can be composed by extending and combining related modules, and discusses how to resolve conflicts in a combined module. Section 5 concludes, also considering some other semantic frameworks that aim to support modularity.

2 Action Semantics

Action semantics uses context-free grammars to define the structure of abstract-syntax trees, and inductively defined semantic functions to give semantics to such trees. Each semantic equation is compositional and maps an abstract-syntax tree to an action representing the meaning of the tree. In this section, we briefly review action notation, and then illustrate how to define, extend, and combine modules through a simple example.

2.1 Action Notation

Actions are semantic entities that represent implementation-independent computational behaviour of programs. The performance of an action, which may be part of an enclosing action, either *completes*, corresponding to a normal termination (the performance of enclosing action continues normally); or *escapes*, corresponding to an exceptional termination (the enclosing action is skipped until a trap is reached); or *fails*, corresponding to abortion of the current alternative (any remaining alternatives should be tried); or *diverges*, corresponding to nontermination (the enclosing action also diverges).

The performance of an action processes transient data; it creates and accesses bindings of token to data; it allocates and manipulates primary storage; and it communicates between distributed agents. Actions have various *facets*. The *basic* facet processes independently of information, focusing on control flow; the *functional* facet processes transient information, including

actions operating on data; the *declarative* facet processes scoped information, including actions operating on bindings; the *imperative* facet processes stable information, including actions operating on storage cells; and the *communicative* facet processes permanent information, including action operating on distributed systems of agents. Actions with different facets can be freely combined to form multi-faceted actions. So-called *yielders* are used in primitive actions to inspect the current information (without changing it).

Some action notation important to understand the underlying concepts of defined languages is explained along with the semantic descriptions below. However, some highly suggestive data notations are not explained for brevity. Note that despite its verbose and casual appearance, action notation is completely formal. A full, formal description of action notation can be found in Mosses' book on action semantics [13], and in a more recent Modular SOS definition [17].

2.2 Language-Definition Modules

A language-definition module consists of two sections: syntax and semantics. The syntax section specifies the syntactic structures (abstract syntax) of the language to be defined; the semantics section specifies the meaning of syntactic structures defined in the syntax section.

In this subsection, we define modules necessary to build an expression language, as an illustration of the general approach. We start with a base module named `Expressions` as follows:

```
Expressions {
  syntax:
    Expr.
  semantics:
    datum >= expressible.
    (*) evaluate: Expr -> action[giving an expressible]
}
```

The syntax section of the `Expressions` module above specifies just the name of a syntactic sort `Expr` (used as a nonterminal symbol in grammars) and nothing else, leaving the detailed syntactic structures open, to be defined by other modules importing the module. The semantics section specifies the name of semantic function with its functionality, leaving its meaning to be defined by other modules importing the module. The functionality of the semantic function `evaluate` indicates that for every abstract-syntax tree `E` in the abstract-syntax domain `Expr`, the semantic entity `evaluate E` is an action which gives an expressible value. The value that can be expressed by an expression is defined to be of sort `expressible`, although what constitutes an expressible value is yet to be defined. The sort inclusion `datum >= expressible` indicates that `expressible` is a subsort of `datum`, which is the sort of all individual items of data processed by actions. The module `Expressions` provides only the basic

notation that all modules involving expressions can share, and thus acts as a base module for various extension modules for expressions.

A module can import other modules. For example, the arithmetic-expression module `Arithmetic Expressions` extending `Expressions` is defined as follows:

```
Arithmetic Expressions {
  import Expressions.
  syntax:
    Expr ::= Num | [[ Expr "+" Expr ]].
    Num  ::= [[ digit+ ]].
    E1,E2:Expr; N:Num.
  semantics:
    expressible >= natural.
    (1) evaluate N = give decimal string-of-characters N.
    (2) evaluate [[ E1 "+" E2 ]] =
          ( evaluate E1 and evaluate E2 ) then
          give the sum of (the given natural#1,
                           the given natural#2).
}
```

The names of imported modules are preceded by the keyword `import`. Then the entire body of each imported module is included in the importing module. The syntax section above specifies an expression to be either a numeral or an expression for addition operation. An alternative `[[` $\cdots$ `]]` specifies trees with the components indicated by $\cdots$. Thus `[[ Expr "+" Expr ]]` represents a tree having three branches: the first in `Expr`, the second a symbol `"+"`, and the third again in `Expr`. As in BNF, the notation `::=` in the grammar rule indicates the inclusion of the specified alternative in the syntax of `Expr`. Each variable representing a syntax tree must be declared with its corresponding syntax domain before its use in the semantics section.

The data notation used to represent the meaning in the semantics section must be specified before its use. In this module, `expressible` is merely required to include `natural`.

A semantic function maps an abstract-syntax tree to an action representing its computational meaning. The semantic equations in action semantics are *compositional*, as in Scott-Strachey style denotational semantics [20,23]: the semantics of any compound phrase is determined in terms of the semantics of its subphrases.

`decimal` in Equation `(1)` is a data operation mapping a string of digits to a natural number. `give` Y is a functional-facet action that gives the value yielded by Y. The meaning of `[[ E1 "+" E2 ]]` can be explained informally as follows: both `E1` and `E2` evaluate to natural numbers, n_1 and n_2, respectively, and then the sum of them is given as a result. Notice how close this informal explanation is to Equation `(2)` above (which is completely formal, despite its "natural" appearance). The action combinator `and` is a basic combinator that represents implementation-dependent order of performance; when there is no

interference between the two sub-actions of `and`, the order of evaluation is not significant. The `and` combinator makes a tuple of transient values given by its sub-actions. The `then` combinator passes transient values from the left sub-action to the right sub-action. `given` *d* yields a transient value given to it, only when it is of sort *d*. Note that `a, the,` and `of` are generally insignificant in action notation, and used only for smoother readability.

The modules for boolean expressions, relational expressions, and conditional expressions can be defined similarly as follows:

```
Boolean Expressions {
  import Expressions.
  syntax:
    Expr ::= "true" | "false" | [[ "not" Expr ]].
    E:Expr.
  semantics:
    expressible >= truth-value.
    (1) evaluate "true" = give true.
    (2) evaluate "false" = give false.
    (3) evaluate [[ "not" E ]] =
          evaluate E then give not the given truth-value.
}

Relational Expressions {
  import Expressions.
  syntax:
    Expr ::= [[ Expr "=" Expr ]].
    E1,E2:Expr.
  semantics:
    expressible >= truth-value.
    (1) evaluate [[ E1 "=" E2 ]] =
          ( evaluate E1 and evaluate E2 ) then
          give (the given natural#1 is the given natural#2).
}

Conditional Expressions {
  import Expressions.
  syntax:
    Expr ::= [[ "if" Expr "then" Expr "else" Expr ]].
    E1,E2,E3:Expr.
  semantics:
    expressible >= truth-value.
    (1) evaluate [[ "if" E1 "then" E2 "else" E3 ]] =
          evaluate E1 then
          ( ( check the given truth-value
              and then evaluate E2 )
            or
```

```
            ( check not the given truth-value
              and then evaluate E3 ) ).
}
```

In Equation (1) of `Conditional Expressions`, `check` Y completes when Y yields true and fails when Y yields false. The action combinator `or` above, generally used for nondeterministic choice, here represents a *deterministic* choice, since at least one of two sub-actions must always fail (the entire action fails when the expressible value given by `evaluate E1` is not a truth-value).

Modules can be combined to specify a programming language. For example, the modules defined so far can be combined to define an expression language as follows:

```
Expression Language {
  import Arithmetic Expressions, Boolean Expressions,
         Relational Expressions, Conditional Expressions.
}
```

The module `Expression Language` above is equivalent (apart from the naming of sub-modules) to the following single module, formed by combining the bodies of the imported modules and collecting together all the alternatives for each (syntactic or semantic) sort:

```
Expression Language {
  syntax:
    Expr ::= Num | [[ Expr "+" Expr ]] | "true" | "false"
           | [[ "not" Expr ]] | [[ Expr "=" Expr ]]
           | [[ "if" Expr "then" Expr "else" Expr ]].
    Num  ::= [[ digit+ ]].
    N:Num; E,E1,E2,E3:Expr.
  semantics:
    datum >= expressible.
    expressible >= natural | truth-value.
    (*) evaluate: Expr -> action[giving an expressible]
    (1) evaluate N = give decimal string-of-characters N.
    (2) evaluate [[ E1 "+" E2 ]] =
          ( evaluate E1 and evaluate E2 ) then
          give the sum of (the given natural#1,
                           the given natural#2).
    (3) evaluate "true" = give true.
    (4) evaluate "false" = give false.
    (5) evaluate [[ "not" E ]] =
          evaluate E then give not the given truth-value.
    (6) evaluate [[ E1 "=" E2 ]] =
          ( evaluate E1 and evaluate E2 ) then
          give (the given natural#1 is the given natural#2).
```

```
    (7) evaluate [[ "if" E1 "then" E2 "else" E3 ]] =
          evaluate E1 then
          ( ( check the given truth-value
              and then evaluate E2 )
            or
            ( check not the given truth-value
              and then evaluate E3 ) ).
}
```

Note that shared notation is the crucial feature when combining modules: their internal structure and import relationship are irrelevant.

3 Extension Modules

The process of giving a name to a program construct is called *binding*. In this section, we define binding-extension modules for expressions. A named expression is called an *expression abstract*, which can be invoked later by simply mentioning its name. We first define base modules for giving names, and then extend the base modules to define the modules for expression bindings, expression blocks, expression parameters, and expressions with effects. On the way, we contemplate the semantics of eager vs. lazy binding, static vs. dynamic scoping, call-by-value vs. call-by-name parameter passing schemes, and expressions with effects.

3.1 Modules for Bindings

According to the Landin-Tennent programming-language design principles [21], the phrases in any semantically meaningful syntactic class may be named, which is specifically called the *abstraction principle*. The base modules for binding (naming) extensions are defined as follows:

```
Identifiers {
  syntax:
    Iden ::= [[ letter (letter | digit)* ]].
  semantics:
    token >= Iden.
}
```

The module `Identifiers` is used to define names. `Iden` is a syntax domain for names, and is specified to be a subsort of `token`, which is the sort of all data items that may be bound to values in actions.

```
Definitions {
  syntax:
    Defn.
  semantics:
    datum >= bindable.
```

```
    (*) elaborate: Defn -> action[producing bindings].
}
```

Definitions (including so-called declarations) are binding constructs, and are grouped into a separate syntax domain, `Defn`, the details of which are yet to be specified. The functionality of the semantic function `elaborate` indicates that for every abstract-syntax tree `D` in the abstract-syntax domain `Defn`, the semantic entity `elaborate D` is an action which produces bindings.

The meaning of an expression abstract may be different depending on *when* the expression is evaluated. An expression may be evaluated before it is bound to a name (called eager binding), or after it is invoked (called lazy binding).

3.1.1 Eager Binding

The module for expression-binding with eager evaluation, `Value Naming`, is defined as follows:

```
Value Naming {
  import Identifiers, Definitions, Expressions.
  syntax:
    Defn ::= [[ "val" Iden "=" Expr ]].
    E:Expr; I:Iden.
  semantics:
    (1) elaborate [[ "val" I "=" E ]] =
          evaluate E then bind I to the given bindable.
}
```

The semantic equation above shows that the body of an expression binding is evaluated before being bound to a name, implying eager evaluation. The sort of all values that can be bound to tokens in actions is called `bindable`, and is yet to be defined. In the case of eager binding, the Landin-Tennent principle motivates identifying `bindable` with `expressible`, or at least letting it include all expressible values; but by leaving the relationship between `bindable` and `expressible` open, the above module may be useful also for composing languages whose design does not dogmatically adhere to the mentioned principle. Note that when not all expressible values are bindable, the above action may fail.

The module `Value Lookup` for name invocation is defined as follows:

```
Value Lookup {
  import Identifiers, Expressions.
  syntax:
    Expr ::= Iden.
    I:Iden.
  semantics:
    (1) evaluate I = give the expressible bound to I.
}
```

The name invocation looks up current bindings and gives the expressible value

bound to the name. Here, the action may fail when not all `bindable` values are `expressible`—just the opposite of the situation with the module `Value Naming`.

3.1.2 Lazy Binding

The module for expression binding with lazy evaluation, `Expression Naming`, is defined as follows:

```
Expression Naming {
  import Identifiers, Expressions, Definitions.
  syntax:
    Defn ::= [[ "fun" Iden "=" Expr ]].
    E:Expr; I:Iden.
  semantics:
    bindable >= abstraction.
    (1) elaborate [[ "fun" I "=" E ]] =
          bind I to the closure abstraction of evaluate E.
}
```

The yielder `abstraction of` A, where A is an action, yields an abstraction encapsulating A, but keeping no bindings. In order to keep the bindings that are current at the evaluation of `abstraction of` A, `closure` must be used: `closure` Y, where Y is a yielder, yields the same abstraction as Y, but when the abstraction is enacted, the encapsulated action receives the bindings that were current at the evaluation of `closure` Y. Thus, `the closure abstraction of evaluate E` yields an abstraction of `evaluate E`, but the abstraction incorporates the bindings available at the time, i.e., at definition-time. Thus, the action `evaluate E` in Equation `(1)` above is not performed, but encapsulated with the current bindings and bound to a name (i.e., the evaluation is delayed). Since an abstraction is bound to a name, abstractions are included to be bindables in the semantics section.

The module `Name Lookup` for lazy invocation is defined as follows:

```
Name Lookup {
  import Identifiers, Expressions.
  syntax:
    Expr ::= Iden.
    I:Iden.
  semantics:
    bindable >= abstraction.
    (1) evaluate I = enact the abstraction bound to I.
}
```

The encapsulated action is enacted and performed, receiving the bindings incorporated at definition-time, when the name is invoked as shown in Equation `(1)` above.

3.1.3 Dynamic Binding

The modules for expression binding with lazy evaluation and dynamic scoping are defined as follows:

```
Dynamic Expression Naming {
  import Identifiers, Expressions, Definitions.
  syntax:
    Defn ::= [[ "fun" Iden "=" Expr ]].
    E:Expr; I:Iden.
  semantics:
    bindable >= abstraction.
    (1) elaborate [[ "fun" I "=" E ]] =
          bind I to the abstraction of evaluate E.
}

Dynamic Name Lookup {
  import Identifiers, Expressions.
  syntax:
    Expr ::= Iden.
    I:Iden.
  semantics:
    (1) evaluate I = enact closure the abstraction bound to I.
}
```

The yielder `closure the abstraction bound to I` yields an abstraction that incorporates the bindings available at the time, i.e., at invocation-time. Thus the enaction performs the action encapsulated in the abstraction, letting it receive the current bindings.

3.1.4 Modules for Multiple Definitions

The module for multiple, simultaneous definitions is defined as follows:

```
Multiple Naming {
  import Definitions.
  syntax:
    Defn ::= [[ Defn "," Defn ]].
    D1,D2:Defn.
  semantics:
    (1) elaborate [[ D1 "," D2 ]] = elaborate D1 and elaborate D2.
}
```

The action combinator `and` implies that the declarations are elaborated independently, and then the disjoint union of the produced bindings is formed (with failure when both declarations produce a binding for the same token).

3.2 Modules for Block Structure

The Landin-Tennent qualification principle says that any semantically meaningful syntactic class may admit local declarations [21]. This means in par-

ticular that any expression belonging to the syntax class `Expr` can have local definitions. A construct admitting local definitions is called a *block*. The module for an expression block is defined as follows.

```
Expression Blocks {
  import Expressions, Definitions.
  syntax:
    Expr ::= [[ "let" Defn "in" Expr ]].
    D:Defn; E:Expr.
  semantics:
    (1) evaluate [[ "let" D "in" E ]] =
          furthermore elaborate D hence evaluate E.
}
```

The action `furthermore elaborate D` overlays the received bindings with the ones produced by `elaborate D`. Then `hence` combinator passes the overlaid bindings to the action `evaluate E`. This corresponds exactly to an ordinary block structure. Thus, bindings declared in `D` can only be referred in the body `E`, not outside the block.

The distinction between *static* and *dynamic* binding arises when an abstract created in the scope of one set of bindings may get applied in the scope of another set. In the examples given above, `"let"` blocks could involve both static and dynamic scopes for expression abstracts: it is the semantics of the abstracts themselves that determines whether or not the abstraction-time bindings get encapsulated together with the action representing the expression evaluation, for later use; and it is the semantics of name lookup that determines whether or not the lookup-time bindings are made available to the action encapsulated in the abstraction. When an abstract has determined that it is using the static bindings, it simply ignores any lookup-time bindings supplied by the semantics of name reference. On the other hand, when the semantics of the abstract ignores the abstraction-time bindings, it depends on the semantics of name lookup as to whether the dynamic bindings will be supplied or not.

3.3 Modules for Parameters

The Landin-Tennent parameterization principle says that phrases from any semantically meaningful syntactic class may be parameters [21]. In particular, any expression belonging to the syntax class `Expr` can be parameters. The following modules are defined to have an expression abstract with an expression parameter, along with an invocation construct; modules dealing with definition abstracts and definition parameters may be specified analogously.

The meaning of an abstraction invocation with an actual parameter can be varied depending on when its parameter (argument) is evaluated. Here we examine the semantics of two different parameter-passing schemes: call-by-value and call-by-name.

3.3.1 Call-by-Value Parameter-Passing

In the call-by-value parameter-passing scheme, an argument is fully evaluated at the time of invocation of a parameterized abstract. Then the evaluated value is bound to a formal parameter. The modules are defined as follows:

```
Expression Naming with Value Parameter {
  import Identifiers, Expressions, Definitions.
  syntax:
    Defn ::= [[ "fun" Iden "(" "val" Iden ")" "=" Expr ]].
    E:Expr; I1,I2:Iden.
  semantics:
    bindable >= abstraction.
    (1) elaborate [[ "fun" I1 "(" "val" I2 ")" "=" E ]] =
          bind I1 to the closure abstraction of
                     ( furthermore bind I2 to the given bindable
                       hence evaluate E ).
}

Call-By-Value {
  import Identifiers, Expressions.
  syntax:
    Expr ::= [[ Iden "(" Expr ")" ]].
    E:Expr; I:Iden.
  semantics:
    bindable >= abstraction.
    (1) evaluate [[ I "(" E ")" ]] =
          ( give the abstraction bound to I
            and
            evaluate E )
          then enact the application of the given abstraction#1
                                      to the given expressible#2.
}
```

Equation `(1)` in the module `Call-By-Value` shows that an argument is evaluated to an expressible, and then the parameterized expression abstract is applied to the expressible, so that the evaluated argument value is bound to a formal parameter when the application is enacted. The binding acts as a local declaration for the body of the abstraction. Equation `(1)` in the module `Expression Naming with Value Parameter` indicates that the bindings current at the definition are used when the parameterized abstraction is applied, implying static scoping. Note that the module `Value Lookup` (rather than `Name Lookup`) should be used together with these modules, which shows that eager binding and call-by-value scheme fit well together.

3.3.2 Call-by-Name Parameter-Passing

In the call-by-name parameter-passing scheme, the evaluation of an argument is delayed until it is used in the body of the called abstract. That is, the

unevaluated argument is bound to a formal parameter, and then evaluated every time the formal parameter is invoked in the body of the called abstract. The modules are defined as follows:

```
Expression Naming with Name Parameters {
  import Identifiers, Expressions, Definitions.
  syntax:
    Defn ::= [[ "fun" Iden "(" "name" Iden ")" "=" Expr ]].
    E:Expr; I1,I2:Iden.
  semantics:
    bindable >= abstraction.
    (1) elaborate [[ "fun" I1 "(" "name" I2 ")" "=" E ]] =
          bind I1 to the closure abstraction of
               ( furthermore bind I2 to the given abstraction
                 hence evaluate E ).
}

Call-By-Name {
  import Identifiers, Expressions.
  syntax:
    Expr ::= [[ Iden "(" Expr ")" ]].
    E:Expr; I:Iden.
  semantics:
    bindable >= abstraction.
    (1) evaluate [[ I "(" E ")" ]] =
          enact the application of the abstraction bound to I
                to the closure abstraction of evaluate E.
}
```

Equation `(1)` in the module `Call-By-Name` shows that an argument `E` of function application is *not* evaluated (i.e., the corresponding action `evaluate E` is encapsulated into an abstraction), and then the parameterized expression abstract is applied to the argument, resulting that the abstraction representing the unevaluated argument is bound to a formal parameter. The `closure` in Equation `(1)` in the module `Call-By-Name` suggests which bindings should be used when the argument is evaluated in the body of parameterized abstract. Note that the abstraction is enacted when it is invoked in the body of parameterized abstract, and thus the module `Expression Lookup` should be used with these modules, showing that lazy binding and call-by-name fit well together.

3.4 Modules for Expression with Effects

In this section, we examine extension modules for adding effects to expressions. We first define a module for an effect-expression extension, `Value References`, with an ML-like syntax, as follows:

```
Value References {
  import Expressions.
  syntax:
    Expr ::= [[ "!" Expr ]]
           | [[ "ref" Expr ]]
           | [[ Expr ":=" Expr ]].
    E,E1,E2:Expr.
  semantics:
    expressible >= cell.
    (1) evaluate [[ "!" E ]] =
          evaluate E then
          give the storable stored in the given cell.
    (2) evaluate [[ "ref" E ]] =
          ( allocate a cell and evaluate E ) then
          ( store the given storable#2 in the given cell#1
            and
            give the given cell#1 ).
    (3) evaluate [[ E1 ":=" E2 ]] =
          ( evaluate E1 and evaluate E2 ) then
          store the given storable#2 in the given cell#1.
}
```

The module `Value References` adds three new constructs—value dereferencing, allocating/storing, and updating—to expressions, as the syntax section shows. Equation `(1)` indicates that an expression `E` evaluates to a cell and a dereferencing operator `!` is used to look up the storable value stored in the cell. Equation `(2)` allocates a cell and evaluates an expression `E`, and then the evaluated storable value is stored in the cell; the allocated cell is returned as a result. In Equation `(3)`, an expression `E1` evaluates to a cell and an expression `E2` evaluates to a storable value; then the storable value is stored in the cell. Note that the expressible values are now extended to include `cell`; the sort `storable`, including all values that can be stored in cells, is, however, left unspecified.

The language module for expression sequencing is defined as follows:

```
Expression Sequencing {
  import Expressions.
  syntax:
    Expr ::= [[ Expr ";" Expr ]].
    E1,E2:Expr.
  semantics:
    (1) evaluate [[ E1 ";" E2 ]] =
          evaluate E1 and then evaluate E2.
}
```

The action combinator `and then` in Equation `(1)` above forces its sub-actions to be performed sequentially.

In this section, we have built extension modules for various language con-

structs: expression bindings, expression blocks, expression parameters and expressions with effects. One could also define declaration bindings, declaration blocks, and declaration parameters in a similar fashion, but the presentation would not illustrate any new points of interest, so we do not bother with that here.

4 Combining Modules

The modules developed in the previous sections can be combined to become complete programming-language modules, as indicated by the module `Expression Language` in Section 2. Combining modules can be achieved simply by importing them together into another module, assuming that the symbols they share correspond to common features. However, we need to consider what should happen when the modules to be combined specify different sort inclusions involving the same sorts, or different semantic equations for the same syntactic constructs.

When there exist two or more different inclusions `s >= s1`, `s >= s2` for the same sort `s` in a combined module, these may be unified into a single inclusion using *sort union* `s1 | s2`, which is supported by the algebraic foundations of action semantics. For instance, in the module `Expression Language` the following sort inclusions:

```
expressible >= natural.
expressible >= truth-value.
```

are unified to give:

```
expressible >= natural | truth-value.
```

Sort union is both commutative and associative, and the way that unions are expressed does not affect the notation used for their values. Mutual inclusions (direct or indirect) between two sorts are no problem either: they merely imply that the sorts are identical, including exactly the same values.

When there are two different semantic equations for the same syntactic construct in the modules to be combined, we say that there is a *conflict*. In fact, some combinations of the modules specified in Section 3 do result in conflicts. Such conflicts may be resolved automatically by unifying the actions in conflicting semantic equations, using the `or` combinator to form a choice between them. To avoid patently undesirable combinations, however, we require that this unification leads to a *deterministic* choice between the alternative actions.

For instance, let us try to combine `Value Lookup` and `Name Lookup`. Then we have a conflict between the following two equations:

```
evaluate I = give the expressible bound to I.
evaluate I = enact the abstraction bound to I.
```

The two actions can be unified provided that the semantic entities of sort

`expressible` do not include any of sort `abstraction`:

```
evaluate I = give the expressible bound to I.
             or
             enact the abstraction bound to I.
```

The action in the new equation above means that if the value bound to `I` is of sort `expressible`, then the expression evaluation must simply give that value; if the value is of sort `abstraction`, then the evaluation must enact the abstraction; and if it is neither an expressible value nor an abstraction, the evaluation must fail.

Such unification of actions permits the modular composition of the languages that, for example, use the same syntax for applying parameterless functions and for referring to ordinary constant values, provided that parameterless functions themselves are not regarded as expressible values. Note, however, that if modules do need to use values of sort `abstraction` in representing expressible values, the abstractions there should always be embedded into other sorts by use of distinct constructor functions, to allow subsequent combination with modules such as `Name Lookup`. Safest of all is to follow the usual practice in action-semantic descriptions, and always embed entities such as abstractions in distinct abstract sorts when using them as (expressible, bindable, or storable) values. For example, a constructor for parameterized functions could embed the sort `abstraction` in an abstract sort `function`, which could then be included in `expressible`, whereas a sort `thunk` embedding the values of parameterless expression abstracts would be excluded from `expressible`.

It appears that `Expression Naming` and `Dynamic Expression Naming` cannot be unified since, as the following equation shows, the unified action would involve an inherently nondeterministic choice:

```
elaborate [[ "fun" I "=" E ]] =
  bind I to the closure abstraction of evaluate E
  or
  bind I to the abstraction of evaluate E.
```

When two modules cannot be unified (deterministically), we say that the two modules are *inconsistent*. Inconsistent modules can however still be combined by *renaming* the syntax of the constructs with the clashing semantic equations. For instance, `Expression Naming` could be combined with the translated module `Dynamic Expression Naming [ [[ "fun" I "=" E ]] |-> [[ "dyn" I "=" E ]] ]`, since the semantic equations in question now concern different syntactic constructs. Such translation allows the combination of modules that are normally used as alternatives; it can also be used merely to adjust the symbols of separately-developed modules to obtain the desired sharing. *Hiding* the syntax of constructs (not illustrated here) gives the effect of removing their semantic equations, after which their semantics may be completely redefined.

Let us look at some modules with no conflict. A first-order lazy functional

language supporting static scoping is defined by including modules as follows:

```
First-Order Lazy Functional Language {
  import Expression Language, Expression Naming, Name Lookup,
         Multiple Naming, Expression Blocks,
         Expression Naming with Name Parameters, Call-By-Name
}
```

Since there are no conflicting semantic equations in the combined modules above, we say that the modules are safely combined. Notice that although `Name Lookup` imports only `Expressions`, which does not itself specify the syntax or semantics of any particular expression constructs, the effect of the combination is just as if all the modules for expression constructs had also been imported.

As an another example, let us look at the module for a first-order eager functional language with effects:

```
First-Order Eager Functional Language with Effects {
  import Expression Language, Value Naming, Value Lookup,
         Multiple Naming, Expression Blocks,
         Expression Naming with Value Parameter, Call-By-Value,
         Value References, Expression Sequencing.
  semantics:
    bindable >= cell.
    storable >= natural.
}
```

The combination in the above module presents no conflicting semantic equations. However, since the sorts `bindable` and `storable` have remained unspecified in the imported modules, they need to be fully specified in the combined module in order for the language to be complete. This means that the decision about which sorts of values should be bindable and/or storable has been left open until we form a complete language. In fact, we might choose them minimally, to be no more than required in the above combined module; at the other extreme, we might choose them as follows:

```
bindable >= cell | natural | truth-value.
storable >= cell | natural | truth-value.
```

Note that in the meanwhile `expressible` in the combined module must include at least the following sorts:

```
expressible >= cell | natural | truth-value.
```

The observant reader may have noticed that the combination of the functional `Expression Language` module with the `Value References` module has undermined the determinism of the described language: assignments occurring in sub-expressions of the same arithmetic expression may be interleaved in any order, and this clearly may lead to different possible values of expressions. Such nondeterminism is quite different from that arising when unifying actions, since it does not involve a choice between performing different actions,

and merely reflects the possibility of observing an internal nondeterminism that was already present in the computational semantics of actions representing expression evaluation; its appearance should therefore not invalidate module combination. Incidentally, even if one does regard nondeterminism due to interleaving as undesirable, and perhaps wishes to prohibit just those programs whose outcome may depend on which interleaving is chosen, the nondeterministic semantics is still needed, in order to distinguish the prohibited programs from the others.

We have illustrated our approach by combining action-semantics modules based on expression-based constructs taken from functional programming languages. It is, of course, equally possible to develop imperative languages by defining a language module for conventional imperative constructs, with a new syntax domain `command` including assignment, loop and sequencing constructs, and then defining and combining extension modules as we have done in this paper.

With the modular structure previously adopted in action semantics, each module typically defines a semantic function on an entire syntactic sort. Thus a conventional action-semantic description of the first-order language with effects would have essentially just two modules defining semantic functions: one for evaluating expressions, the other for elaborating definitions. The inherent modularity of action notation allows reuse of individual semantic equations in other descriptions, but it is unlikely that such large modules would ever be reused *in toto*, merely by referring to them.

In contrast, the finer modular structure produced by combining modules, as illustrated in this section, should encourage the direct reuse and combination of modules from one action-semantic description in later ones. Note however that although the *ad hoc* meta-notation used in this paper is quite convenient and perspicuous, it could probably be improved, e.g. to avoid the repetitive variable declarations.

Finally, let us note that we have here focussed on the modules defining semantic functions. In large-scale action-semantic descriptions, also the specification of *semantic entities* (providing data types and action abbreviations for higher-level concepts, such as compound variables and communication protocols) has an interesting modular structure. When the modules defining semantic functions are structured so as to group related constructs together, the modules defining semantic entities may be more easily localized, making it apparent that they are only needed in connection with the semantics of particular language constructs. However, an appropriate visualization of the import relationship between modules may be as effective as (and more flexible than) an explicit indication of the intended grouping of modules for semantic entities with those for semantic functions.

5 Conclusion

We have demonstrated how to use action semantics to define, extend, and combine language modules. The good modularity and extensibility of action semantics help us systematically develop programming languages. Particularly in the presence of the language modules for similar languages, new language modules can be defined with only a few modifications when using the fine-grained modular structure proposed here.

The ideas developed in this article can be adopted to guide the design of domain-specific languages and/or rapidly prototyped special-purpose languages. When one designs such a language, one can analyze its problem domain, design a core language, and then gradually build up language features to it according to some rational design principles, such as the Landin-Tennent principles.

It should also be possible to reuse modules in the style of those shown in this paper when describing previously-designed languages; in a future paper, we intend to report on a case-study involving the action-semantic description of an existing domain-specific language. However, *tool support* is clearly essential for checking the well-formedness and consistency of modules (and ultimately, for generating prototype implementations from them). It appears that it would be feasible to map our modules to ASF+SDF modules using the Meta-Environment [2], as in the ASD Tools developed by van Deursen and Mosses [3]. The implementation of such tools should be facilitated by adopting the much-simplified version of action notation that was proposed by Lassen, Mosses, and Watt at the Action Semantics Workshop last year [10], and also by the forthcoming incorporation of SDF2 in the Meta-Environment.

Other semantic frameworks that aim to provide a high degree of modularity have been developed. For example, Moggi [12] has proposed the use of monads and monad transformers in denotational semantics. Cartwright and Felleisen [1] have proposed the use of an operationally-motivated style of denotational semantics, in the interests of modularity; it uses auxiliary notation similar to that of Moggi, but appears to be not so general (nondeterminism and concurrency are excluded). Liang and Hudak have implemented Moggi's monad transformers [12] in their *modular monadic semantics* framework [11]. Wansbrough and Hamer have used the modular monadic framework to give a modular monadic semantics of action notation, and called it *modular monadic action semantics* [24]. In response to such work, Mosses has developed *Modular SOS* [16] which provides significantly greater modularity in SOS, and he has redefined action notation in Modular SOS, improving dramatically the modularity of the definition [17]. The Abstract State Machine (ASM) approach [6] also provides modularity for operational semantics, through the use of a particularly neat notation for updating and accessing components of a global configuration. Although the ASM approach generally lacks the compositionality of SOS-based frameworks, the Montages presentation of ASM

[9] provides a separate module for each syntactic construct, specifying how a local ASM for that construct is plugged into the global ASM, together with the firing rules associated with the local ASM.

It appears that none of the above frameworks supports the definition and combination of language modules to the same extent as action semantics does. For instance, the monadic approach to denotational semantics would require a particular composition of monad transformers to be specified for each module; not only would this be repetitive, but also it is unclear how to combine independently-specified compositions. Similarly, Modular SOS requires the specification of compositions of so-called label transformers—although there it would be quite straightforward to combine the transformers (since their order of composition is insignificant, and their symbolic indices allow duplication to be avoided). The Montages approach to ASM inspired our reconsideration of the modular structure of action semantics; it supports combination of modules for individual constructs into a complete language, but to our knowledge it does not allow modules that group related constructs together, which seems to be crucial for factorization and abbreviation of descriptions. Moreover, it seems that Montages modules from one language description cannot in general be reused without reformulation in other descriptions, since the notation used in the underlying ASM formalism may well vary.

There is no silver bullet in designing a good programming language. However, we hope a design tool such as the one proposed in this article may be used to enhance the quality of language-design activity.

Acknowledgement

David Schmidt provided the idea of defining and combining language-definition modules; Olivier Danvy encouraged us to write this paper and made numerous comments on the contents and structure of the paper; the anonymous referees supplied useful comments and corrections.

References

[1] Cartwright, R. and M. Felleisen, *Extensible denotational language specifications*, in: M. Hagiya and J. C. Mitchell, editors, *TACS'94, Symposium on Theoretical Aspects of Computer Software, Sendai, Japan*, Lecture Notes in Computer Science **789** (1994), pp. 244–272.

[2] van Deursen, A., J. Heering and P. Klint, editors, "Language Prototyping," AMAST Series in Computing **5**, World Scientific, 1996.

[3] van Deursen, A. and P. D. Mosses, *ASD: The action semantic description tools*, in: *AMAST'96, Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology, Munich*, Lecture Notes in Computer Science **1101** (1996), pp. 579–582.

[4] Doh, K.-G. and D. A. Schmidt, *Extraction of strong typing laws from action semantics definitions*, in: *ESOP'92, Proc. European Symposium on Programming, Rennes*, Lecture Notes in Computer Science **582** (1992), pp. 151–166.

[5] Doh, K.-G. and D. A. Schmidt, *Action semantics-directed prototyping*, Computer Languages **19** (1993), pp. 213–233.

[6] Gurevich, Y., *Evolving algebras 1993: Lipari guide*in: E. Börger, editor, *Specification and Validation Methods*, Oxford University Press, 1995 .

[7] Hoare, C. A. R., *Hints on programming language design*, in: *POPL'73, Proc. 1st ACM Symp. on Principles of Programming Languages* (1973), also in [8].

[8] Hoare, C. A. R. and C. B. Jones, "Essays in Computing Science," Prentice-Hall, 1989.

[9] Kutter, P. and A. Pierantonio, *Montages: Specifications of realistic programming languages*, JUCS **3** (1997), pp. 416–442.

[10] Lassen, S. B., P. D. Mosses and D. A. Watt, *An introduction to AN-2, the proposed new version of Action Notation*, in: *AS 2000*, number NS-00-6 in Notes Series, BRICS, Dept. of Computer Science, Univ. of Aarhus, 2000, pp. 19–36.

[11] Liang, S. and P. Hudak, *Modular denotational semantics for compiler construction*, in: *ESOP'96, Proc. 6th European Symp. on Programming, Linköping*, Lecture Notes in Computer Science **1058** (1996), pp. 219–234.

[12] Moggi, E., *An abstract view of programming languages*, Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh (1990).

[13] Mosses, P. D., "Action Semantics," Number 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[14] Mosses, P. D., *Theory and practice of action semantics*, in: *MFCS'96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science, Cracow, Poland*, Lecture Notes in Computer Science **1113** (1996), pp. 37–61.

[15] Mosses, P. D., *A tutorial on action semantics* (1996), tutorial notes for FME'94 (Formal Methods Europe, Barcelona, 1994) and FME'96 (Formal Methods Europe, Oxford, 1996), also available from the author at `http://www.brics.dk/Projects/AS/`.

[16] Mosses, P. D., *Foundations of Modular SOS (extended abstract)*, in: *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska Poreba, Poland*, Lecture Notes in Computer Science (1999), pp. 70–80, full version published as BRICS RS-99-54, Dept. of Computer Science, University of Aarhus, 1999.

[17] Mosses, P. D., *A modular SOS for action notation*, Technical Report BRICS RS-99-56, Dept. of Computer Science, University of Aarhus (1999).

[18] Mosses, P. D. and D. A. Watt, *Pascal: Action semantics* (1993), draft, Version 0.6, Available from the authors at `http://www.brics.dk/Projects/AS/`.

[19] Plotkin, G. D., *A structural approach to operational semantics*, Lecture Notes DAIMI FN-19, Dept. of Computer Science, University of Aarhus (1981).

[20] Schmidt, D. A., "Denotational Semantics: A Methodology for Language Development," Allyn and Bacon, 1986.

[21] Schmidt, D. A., "The Structure of Typed Programming Languages," MIT Press, 1994.

[22] Schmidt, D. A., *Personal communications* (1999).

[23] Stoy, J. E., "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory," MIT Press, 1977.

[24] Wansbrough, K. and J. Hamer, *A modular monadic action semantics*, in: *Proc. Conf. on Domain-Specific Languages* (1997), pp. 157–170.

[25] Watt, D. A., "Programming Language Syntax and Semantics," Prentice-Hall, 1991.

[26] Watt, D. A., *Standard ML action semantics* (1997), draft, Version 0.5, Available from the author at `http://www.brics.dk/Projects/AS/`.

[27] Watt, D. A., *The static and dynamic semantics of Standard ML*, in: *AS'99, 2nd International Workshop on Action Semantics (ed. Mosses, P.D., and Watt, D.A.)*, BRICS NS-99-3, Dept. of Computer Science, University of Aarhus, 1999, pp. 155–172.